Linux Heap TCache Poisoning

Dr Silvio Cesare

InfoSect

Summary

In this paper, I introduce the reader to a heap metadata corruption against the current Linux Heap Allocator, ptmalloc. The attack is performed via corrupting, or poisoning the tcache such that malloc returns an arbitrary pointer. This may allow for control flow hijacking if malloc returns a pointer to a function pointer and an attacker is able to write to that malloc returned buffer. TCache poisoning is possible from heap corruption including buffer overflows and Use-After-Frees.

Introduction

In July 2000, an article on exploiting a Nestcape Navigator heap overflow was released https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability and heap metadata corruption was born. The generic attack against the Linux heap could turn the free of a buffer that had a string-based buffer overflow, into an arbitrary write to memory, known as a write-what-where primitive.

Writing what you want to where you want in memory is a powerful primitive. Attackers could leverage this write-what-where to hijack control flow. They could do this by overwriting functions pointers present in the process image, such as those in the Global Offset Table (GOT) function pointers that get filled in by the dynamic linker when symbol resolution occurs. Today hijacking via the GOT has been mitigated against by having these entries remapped as read-only at load time. However, there are still other function pointers still available in a process image. For example, malloc hooks.

The metadata corruptions attacks of yester year took advantage of unlinking a node in a linked list during free chunk coalescing. The unlink operation used pointer operations in the style of node->next->prev = node->prev. If an attacker was able to control the next and prev pointers in a node, they could turn this pointer dereference and assignment into a write-what-where.

Today, most allocators have been hardened using a simple integrity check on the linked list that prevent such a power write-what-where primitive from being available. However, heap allocators have other failure cases that are potentially useful to an attacker. Thus, if attackers are able to obtain any of the following results from the heap allocator, they gain an advantage:

- Returning an arbitrary pointer from malloc
- Returning a near arbitrary pointer from malloc (such as to a part of the stack)
- Returning the same allocated memory twice
- Allocating memory that overlaps with another allocation

Returning an arbitrary pointer from malloc is a powerful primitive. If attacker controlled input fills an object, then an attacker makes the allocation for that object point to somewhere useful, such as a function pointer. Then the attacker is able to fill in the object details since the program believes the allocation is sound. Instead, an attacker might be able to overwrite a function pointer to begin a ROP chain or gain code execution.

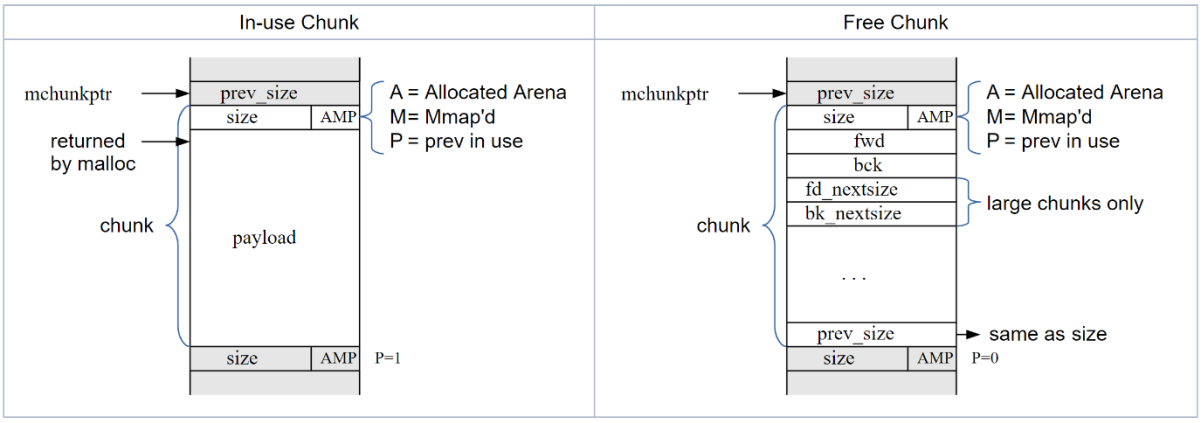
The same style of attack exists for other primitives such as overlapping allocations or having the same memory returned twice from the allocator. If the attacker writes to one object the program

thinks is safe, but instead overwrites the function pointer of another object, code execution may be possible.

The Linux Heap Allocator

The Linux heap allocator uses ptmalloc internally. Ptmalloc is derived from Doug Lea's Malloc. An important note is that inline metadata surrounds chunks of free and allocated memory. This inline metadata can be corrupted and course the allocator to behave in ways useful to an attacker, triggering the modern primitives I discusses earlier.

Here is what the documentation shows, taken from
https://sourceware.org/glibc/wiki/MallocInternals



Malloc chunks are core structures but a modern allocator also consists of other heap metadata. Arenas are a heap structure that reduces lock contention in multi-threaded environments. Different threads may be associated with different arenas, and each arena holds free and in-use malloc chunks.

Another concept the ptmalloc allocator uses is "bins". Bins are freelists that hold linked lists of free chunks of memory. Thus, when an allocation happens, these bins are examined to determine if any free chunks are present. If they are, the chunks can be removed from the bins and returned to the user who requested the allocation. Naturally, the chunk that is returned is also surrounded by heap metadata. The user is not to access this metadata, but nevertheless, it surrounds the chunk.

Bins are also divided into different types. There are 4 bins – fast, small, large, and unsorted – not including the tcache to be documented later on. These bins are for different purposes and also hold chunks of different sizes appropriate for the specific bin.

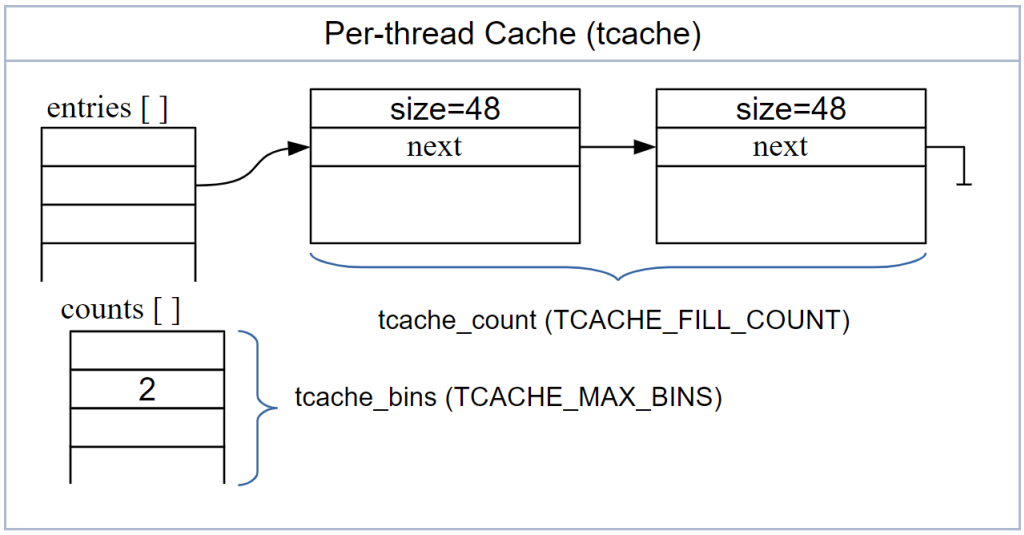
If a bin cannot service an allocation, then it's still possible to give memory back to the user if the heap has available memory, or the allocator may request more memory from the underlying OS and extend the heap.

TCache

The thread cache, or tcache, is an optimization improvement to ptmalloc that was introduced in glibc 2.26. Ubuntu 18.04LTS is one OS distribution that uses the tcache. The motivation of the tcache is that it gives the ability for individual threads to access a cache of free chunks without competing for an arena lock. The tcache is similar to the fastbins but with no arena lock contention.

The tcache is similar to other bins such as the fastbins and as such includes freelists with malloc chunks of the same bucket size going into each freelist. The tcache has a limit on the size of each of these freelists and is currently hardcoded to a maximum of 7 chunks. Like fastbins, there is no coalescing or consolidating of free chunks. The tcache is implemented as a singly linked list similar to fastbins.

Let's examine the data structure for a tcache entry taken from https://sourceware.org/glibc/wiki/MallocInternals.



Importantly, the forward (next) pointer uses in the tcache freelists is placed in the original chunk payload area when the chunk was originally allocated.

That the forward (next) pointer is in the payload of the freed chunk, this allows the possibility of use-after-frees (UAF). At a minimum, allowing information disclosure of heap addresses if memory reads are performed on these chunks.

The tcache is examined firstly during _int_malloc and _int_free in malloc.c, thus giving this optimization, every opportunity to be utilised. Moreover, on occasion, other bins such as fastbins may spill chunks onto the tcache for more opportunity for tcache utilization.

The tcache operates on a Last In First Out (LIFO) basis. When chunks are freed, they try to get placed onto the tcache so that they can be recycled when an allocation occurs.

When a free chunk is placed on the tcache, it is placed at the head of the list. When an item is removed from the tcache, it is removed from the head of the list.

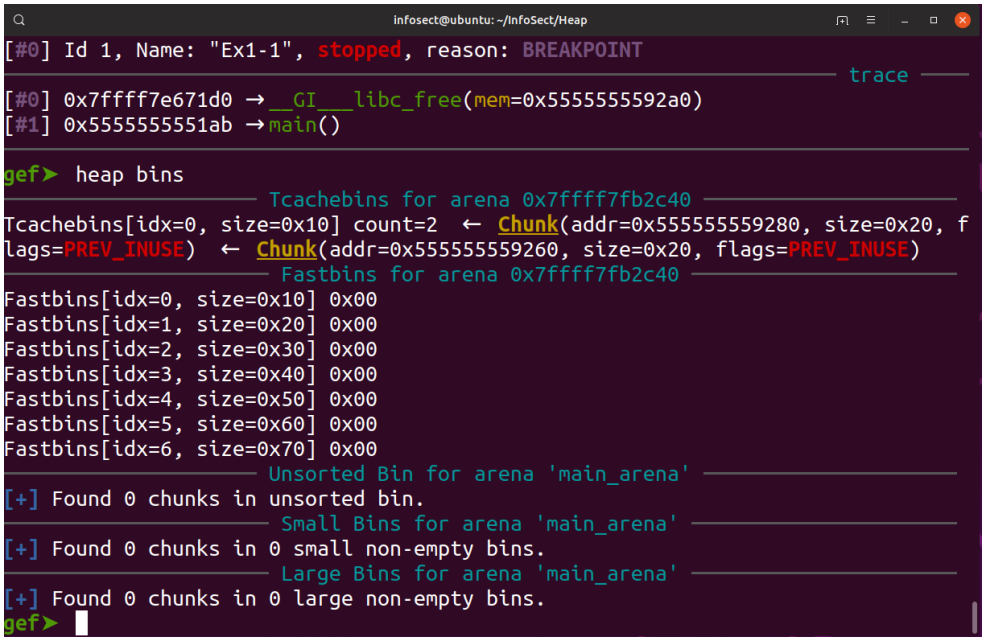
Let's look at allocating 3 small buffers of the same size and then freeing those 3 buffers. Our program looks like this:

```
infosect@ubuntu: ~/InfoSect/Heap
#include <stdio.h>
#include <stdlib.h>

int
main()
{
        long *a, *b, *c;

        a = malloc(8);
        b = malloc(8);
        c = malloc(8);
        free(a);
        free(b);
        free(c);

        exit(0);
}
~
~
~
~
~
~
"Ex1-1.c" 17 lines, 163 characters
```

Let's set a breakpoint after the first free. The first free places the chunk into the tcache. We can see this by examining the ptmalloc heap structure with the GDB plugin GEF. Note that our chunk is at address 0x55..59260 (abbreviated for clarity) and the (internal) size is 0x20.

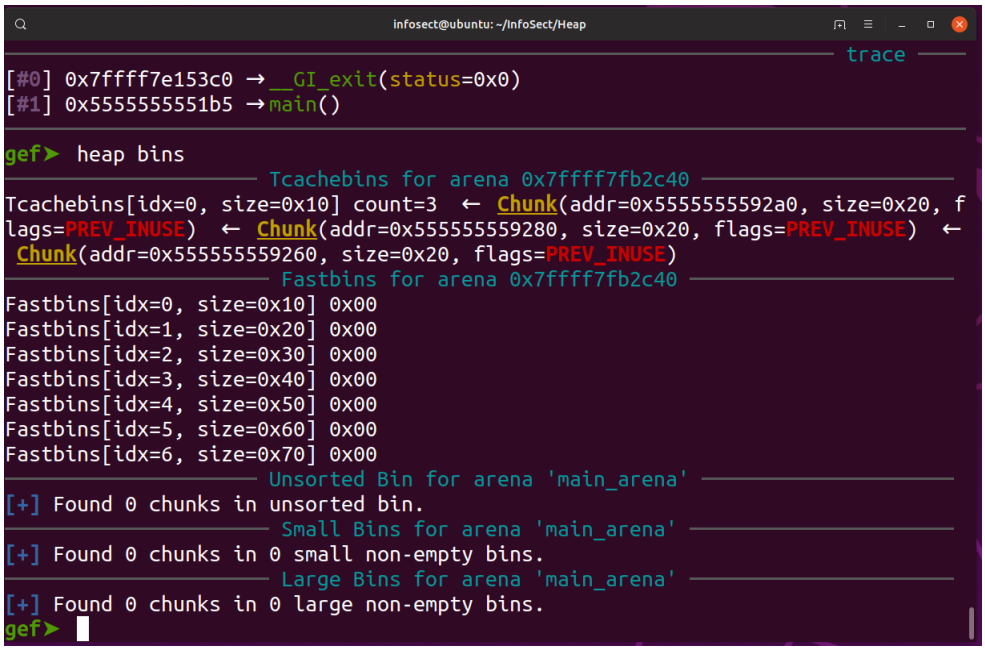
```
infosect@ubuntu: ~/InfoSect/Heap
[#0] Id 1, Name: "Ex1-1", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7ffff7e671d0 → __GI___libc_free(mem=0x555555559280)
[#1] 0x55555555519f →main()
──────────────────────────────────────────────────────────────────────────────────

gef➤  heap bins
──────────────────── Tcachebins for arena 0x7ffff7fb2c40 ────────────────────
Tcachebins[idx=0, size=0x10] count=1  ← Chunk(addr=0x555555559260, size=0x20, f
lags=PREV_INUSE)
──────────────────── Fastbins for arena 0x7ffff7fb2c40 ─────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
──────────────────── Unsorted Bin for arena 'main_arena' ────────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────── Small Bins for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────── Large Bins for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤
```

Let's break after the next free and examine the bins again.



We can see that a 2nd entry has been added. The new chunk is at address 0x55..59280 and is placed at the head of the list. The chunk we freed earlier is now in the 2nd position.

Let's break after the final free and examine the bins again.



We can see our 3rd free chunk at address 0x55..592a0 has been placed at the head of the list.

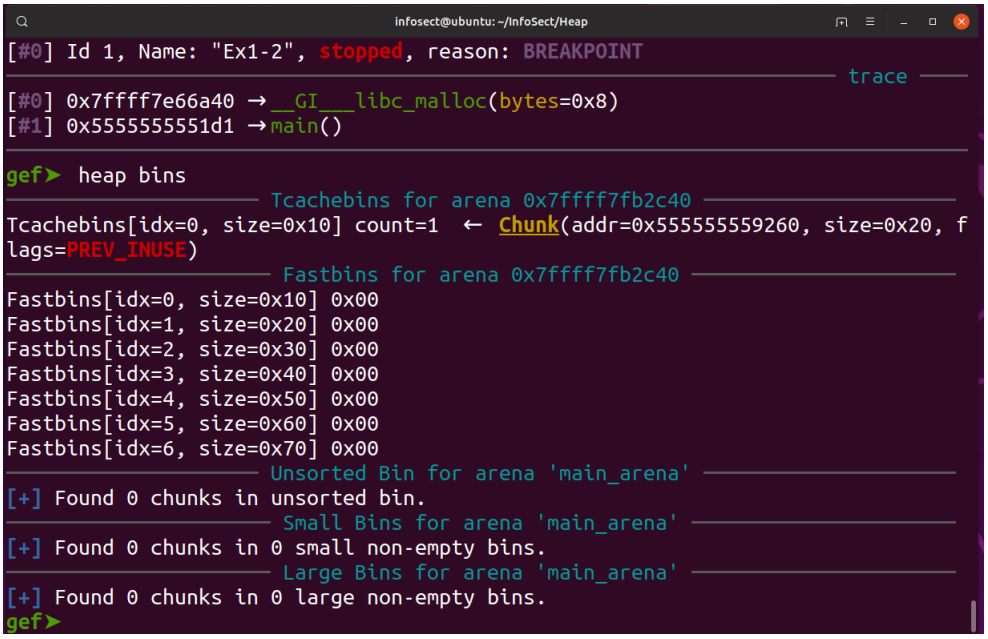
Now let's modify our program so that we do 3 additional allocations of the same size after we have freed our buffers. Our program now looks like this:

```
infosect@ubuntu: ~/InfoSect/Heap
#include <stdio.h>
#include <stdlib.h>

int
main()
{
        long *a, *b, *c;

        a = malloc(8);
        b = malloc(8);
        c = malloc(8);
        free(a);
        free(b);
        free(c);
        a = malloc(8);
        b = malloc(8);
        c = malloc(8);

        exit(0);
}
~
~
~
"Ex1-2.c" 20 lines, 211 characters written
```

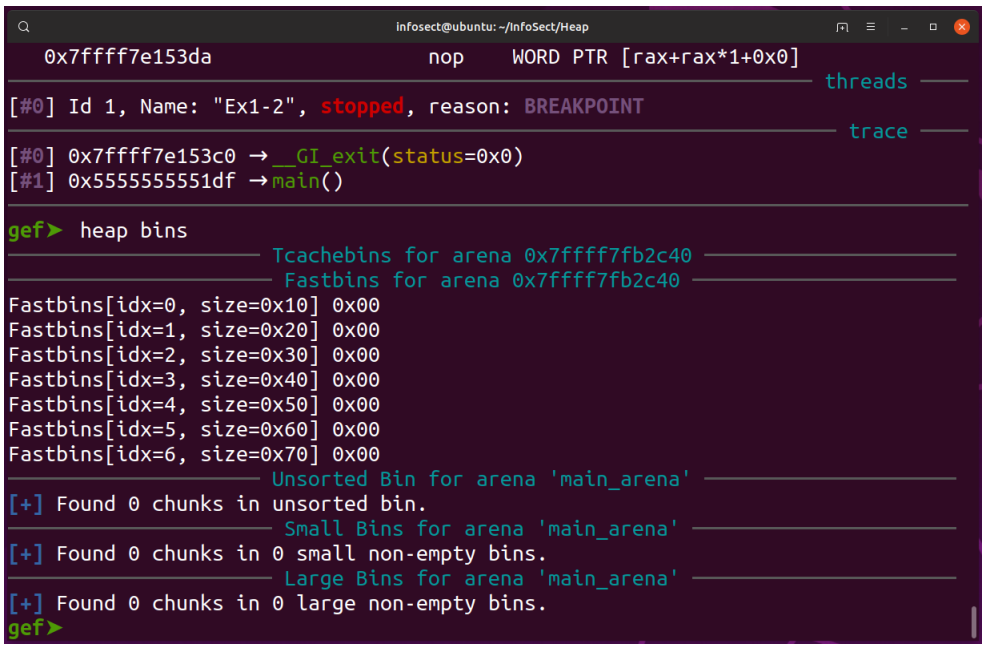
Even though this is a different program, our first 3 allocations gave us the same locations in memory as the previous program. This is in part because we aren't using Address Space Layout Randomization (ASLR). We should assume that before our malloc, then the tcache looks like it did earlier after freeing 3 chunks. Let's set a new breakpoint after the frees and after the first additional malloc.

```
infosect@ubuntu: ~/InfoSect/Heap
[#0] Id 1, Name: "Ex1-2", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7ffff7e66a40 → __GI___libc_malloc(bytes=0x8)
[#1] 0x5555555551c3 → main()
──────────────────────────────────────────────────────────────────────────────────────────
gef➤  heap bins
──────────────────── Tcachebins for arena 0x7ffff7fb2c40 ────────────────────
Tcachebins[idx=0, size=0x10] count=2  ← Chunk(addr=0x555555559280, size=0x20, f
lags=PREV_INUSE)  ← Chunk(addr=0x555555559260, size=0x20, flags=PREV_INUSE)
──────────────────── Fastbins for arena 0x7ffff7fb2c40 ─────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
──────────────────── Unsorted Bin for arena 'main_arena' ────────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────── Small Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────── Large Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤
```

The head of the tcache for that index has been removed and is what our new allocation returns. Let's look after the next allocation.



As expected, the LIFO tcache freelist pops the head of the list and uses for the new allocation. Let's look after the final allocation.



And the tcache is empty again.

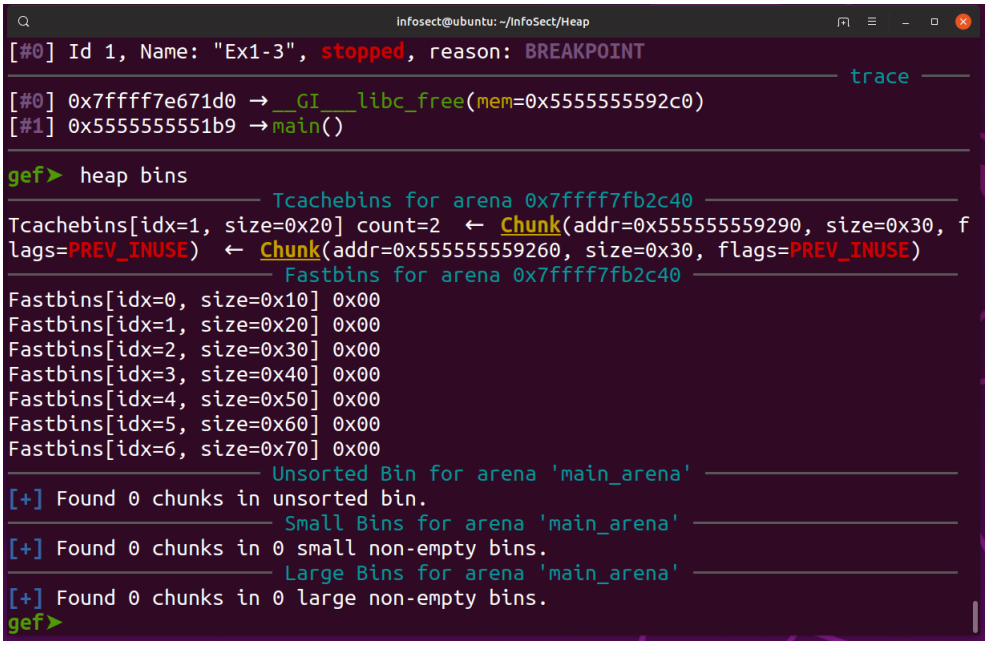
Remember the tcache has different freelists for different sized chunks. Let's look when we mix sizes in our buffers. This is our new program:

```
infosect@ubuntu: ~/InfoSect/Heap
#include <stdio.h>
#include <stdlib.h>

int
main()
{
        long *a, *b, *c, *d;

        a = malloc(32);
        b = malloc(32);
        c = malloc(8);
        d = malloc(8);
        free(a);
        free(b);
        free(c);
        free(d);

        exit(0);
}
~
~
~
~
"Ex1-3.c" 19 lines, 195 characters written
```

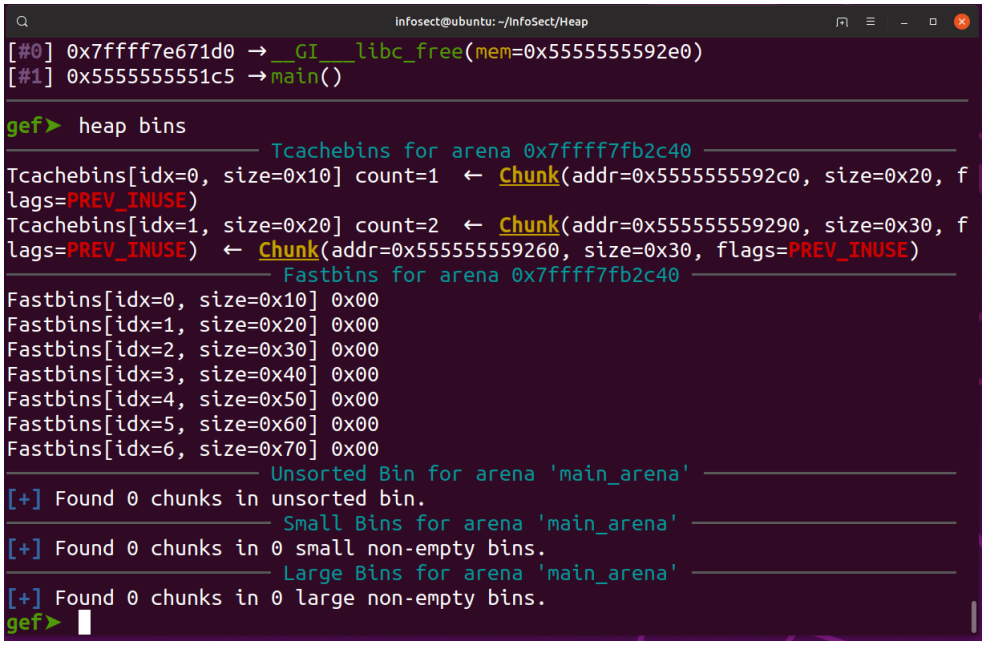
Let’s set up breakpoints after every deallocation.

```
infosect@ubuntu: ~/InfoSect/Heap
[#0] Id 1, Name: "Ex1-3", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7ffff7e671d0 →__GI___libc_free(mem=0x555555559290)
[#1] 0x5555555551ad →main()
──────────────────────────────────────────────────────────────────────────────────────
gef➤  heap bins
──────────────────── Tcachebins for arena 0x7ffff7fb2c40 ────────────────────
Tcachebins[idx=1, size=0x20] count=1  ← Chunk(addr=0x555555559260, size=0x30, f
lags=PREV_INUSE)
──────────────────── Fastbins for arena 0x7ffff7fb2c40 ─────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
──────────────────── Unsorted Bin for arena 'main_arena' ────────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────── Small Bins for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────── Large Bins for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤
```

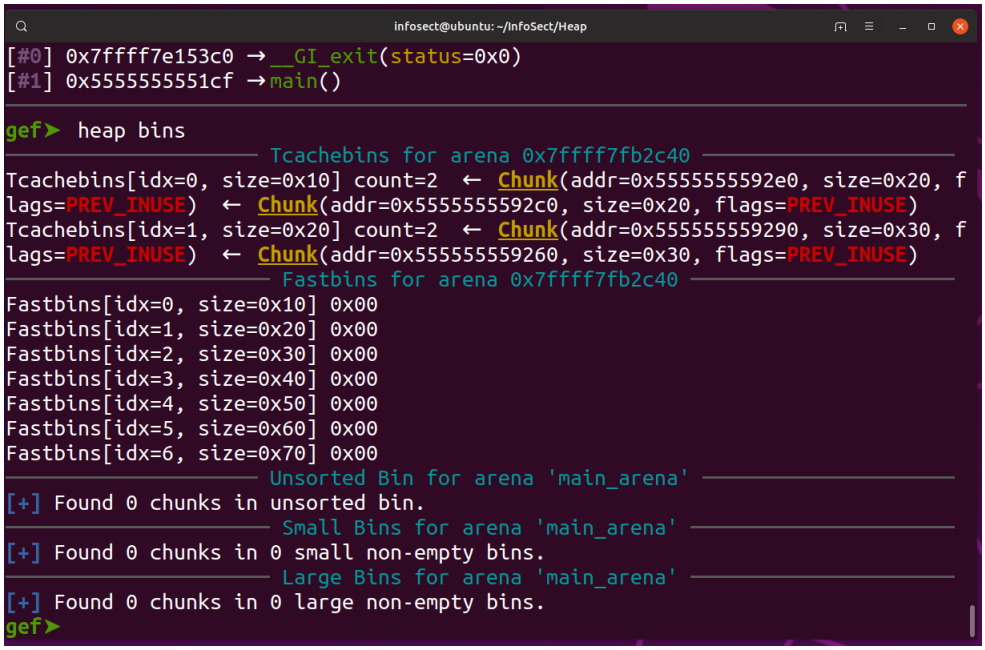
Here we see the deallocation of the 32 byte buffer goes into index 1.



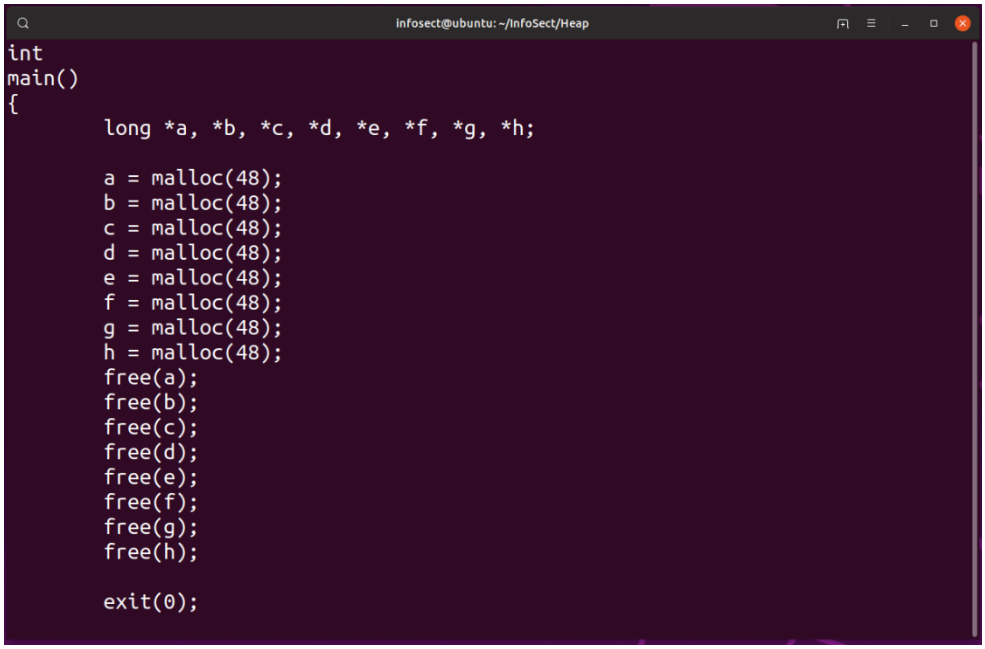
And the next free goes to the head of the list in the same index. Now what happens when we break after the free of 8 bytes?



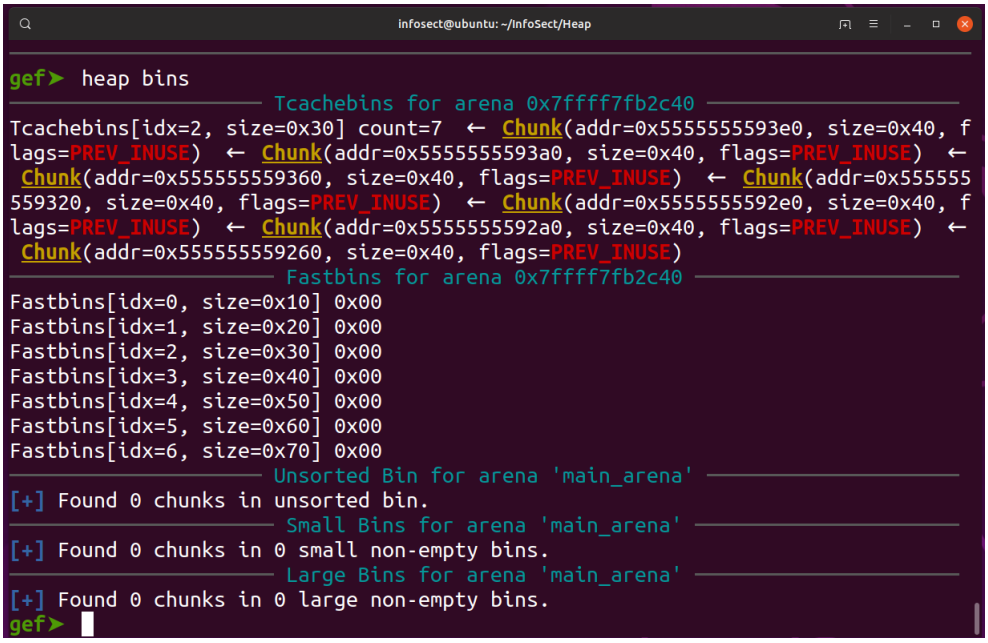
We can see that because the size of the chunk is different, a different freelist is used which holds buckets for the appropriate range of chunk sizes. In this case, for an 8 byte buffer, it goes into index 0. Let's break after the final deallocation of the last 8 byte buffer.



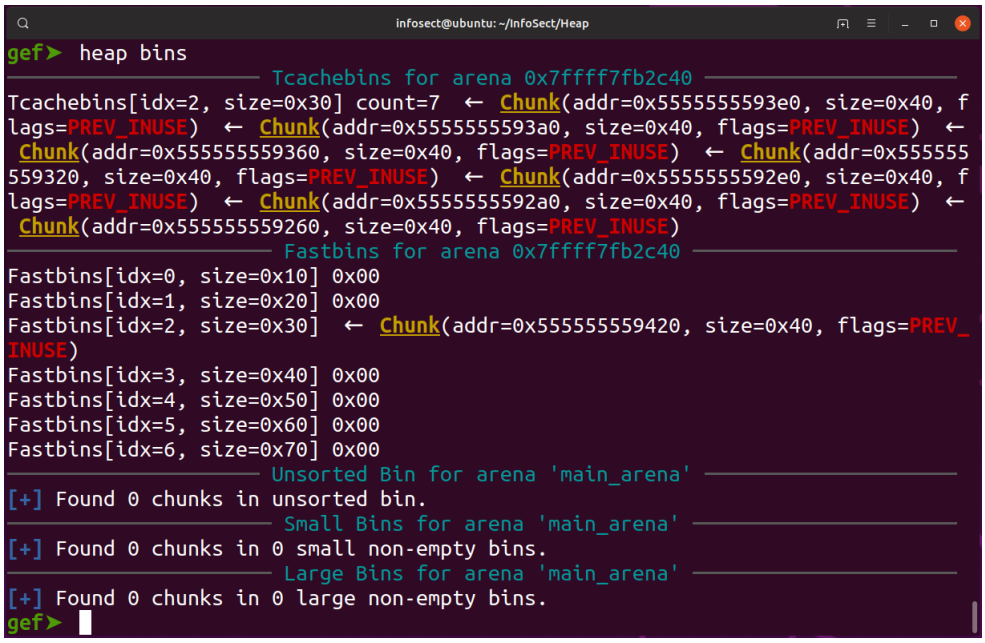
What happens when the tcache freelist becomes full? Then other bins must be used. Here is a new program that does 8 allocations and 8 frees. Note that the tcache bins can only hold 7 entries at a time.



Let's set a breakpoint after the 7th free when the tcache bin becomes full and then on the following free.



The buffer size was 48 which meant the tcache used index 2 to store these chunks. Now there are 7 chunks in the tcache bin/freelist. Let's look after the next free.



We can see that the tcache did not change and instead a fastbin was used to hold the free chunk. The allocator uses different strategies for where to put these free chunks, but if the size is ok and other checks pass, then the fastbin is used as above.

TCache Poisoning

In this document, I will document a single attack against the ptmalloc heap allocator. There are in fact numerous attacks against the allocator, but I will only go in depth on tcache poisoning.

In the tcache poisoning attack, the heap allocator returns an arbitrary attacker-chosen pointer from malloc. This can be achieved by manipulation of a tcache freelist.

Remember that the tcache is a singly linked list joining free chunks together. If an attacker were to manipulate this linked list such that then they could modify a node in this freelist such that it was linked to a ‘fake’ chunk whose position is what the attacker wanted malloc to return.

In fact, this attack is not dependent on many requirements. There is little to no integrity checking on where nodes are connected to, and there are no other modifications to memory that need to be made except modifying this forward pointer in the node belonging to the appropriate tcache freelist.

We also have to remember that the tcache is divided into bins (freelists) of different sizes. When a chunk is freed or allocated, it is associated with a particular index in the tcache. This index and the associated freelist have chunks of all of the same bucket size.

Thus, if a freelist in the tcache is corrupted, then the chunk size is freed must be of the appropriate size to go into the correct bin. And when a chunk if allocated, the correct size must be requested to utilize the corrupted freelist.

One the tcache entry is poisoned, it must be returned from malloc. To do this, the attacker must allocate a suitable number of chunks of the right size to teach the corrupted node. The attacker must then allocate one more buffer to return the corrupted forward pointer. At this point, the attacker obtains the arbitrary pointer from malloc and thus might aim to overwrite a function pointer elsewhere in the program to hijack control flow.

There are a number of ways the tcache can be poisoned. Either a Use-After-Free or a heap overflow is suitable.

In a Use-After-Free bug, the attacker is required to be able to write to the payload of the freed chunk after it is freed. The attacker only needs to write a single pointer at the beginning of the payload as this corresponds to the forward pointer of the tcache structure.

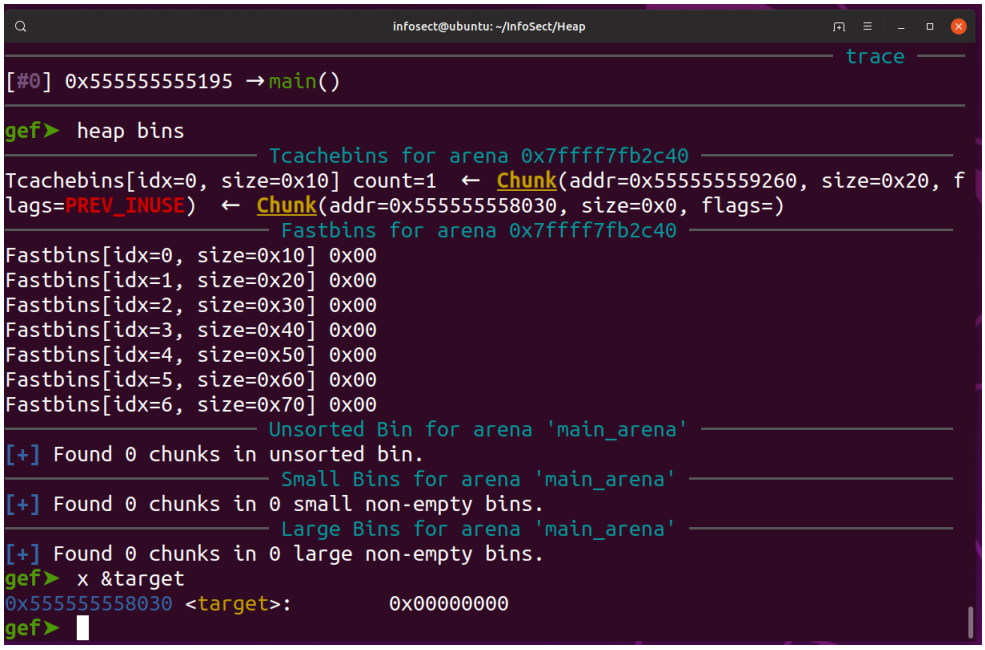
Here is small program that puts a chunk in the tcache and then poisons it before getting malloc to return an arbitrary pointer. The goal of this program is to take a ‘target’ variable and be able to overwrite it with arbitrary attacker controlled input. The way to do this in the case below is to have malloc return the address of the target variable.

```
infosect@ubuntu: ~/InfoSect/Heap
#include <stdio.h>
#include <stdlib.h>

static long target = 0;

int
main()
{
        long *a, *recycled_a, *b;

        a = malloc(8);
        // create chunk in tcache
        free(a);

        // poison foward pointer
        *a = (long)&target;

        // recycle chunk from tcache
        recycled_a = malloc(8);

        // return poisoned pointer
        b = malloc(8);

        // overwrite target
        *b = 0x4141414142424242;

        fprintf(stderr, "%lx\n", target);
        exit(0);
}
~
~
```

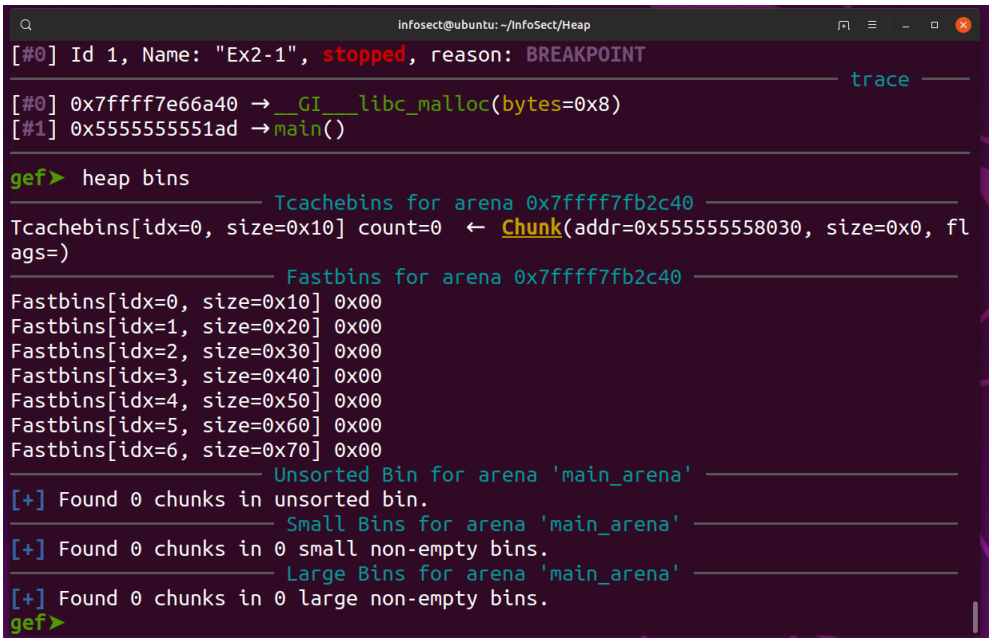
Let’s set a breakpoint after the first free.

```
infosect@ubuntu: ~/InfoSect/Heap
──────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "Ex2-1", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x555555555187 →main()
──────────────────────────────────────────────────────────────────────────────────
gef➤  heap bins
──────────────────── Tcachebins for arena 0x7ffff7fb2c40 ────────────────────
Tcachebins[idx=0, size=0x10] count=1  ← Chunk(addr=0x555555559260, size=0x20, f
lags=PREV_INUSE)
──────────────────── Fastbins for arena 0x7ffff7fb2c40 ─────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
──────────────────── Unsorted Bin for arena 'main_arena' ────────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────── Small Bins for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────── Large Bins for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤
```

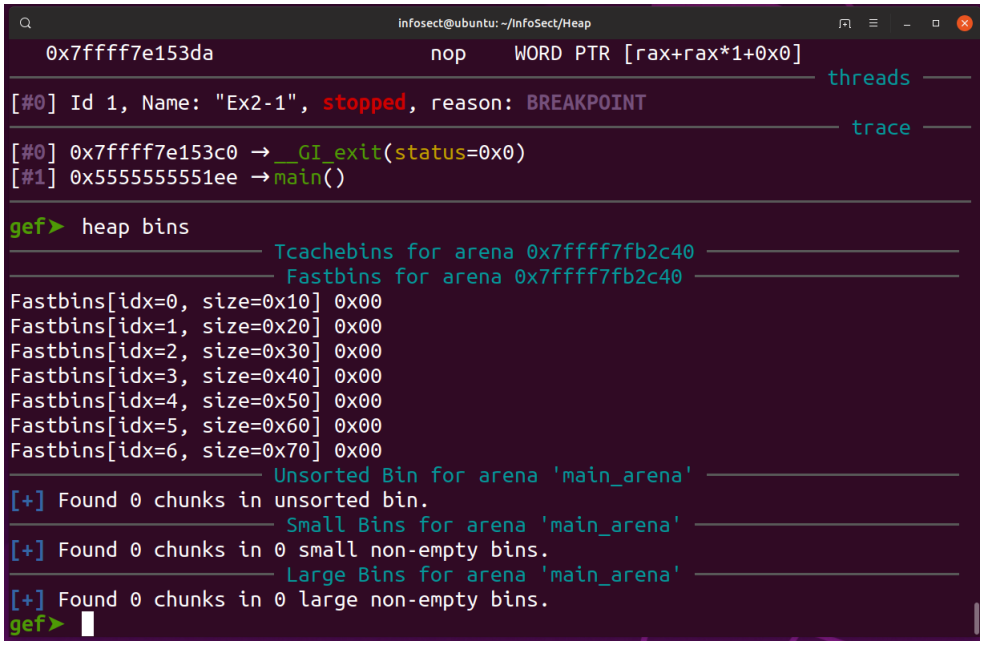
We can see that there is an entry in the tcache associated with the buffer we freed. Now let’s look at what happens when we poison that tcache node’s forward pointer with the address of the target - *a = (long *)&target. For this part, we have our target pointer at 0x55..58030.



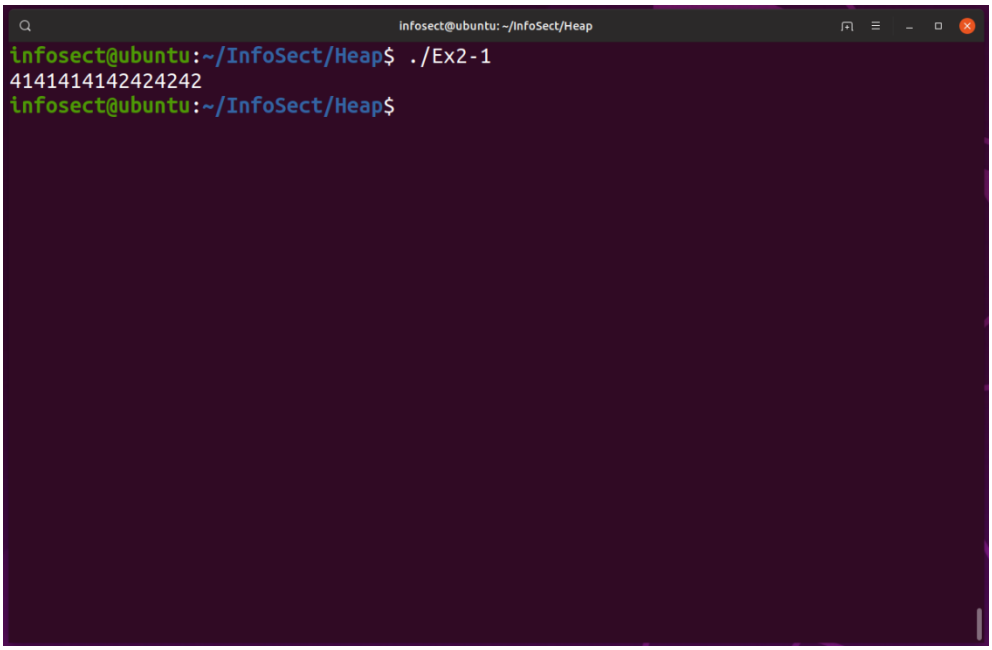
And we can see now that by writing to the freed chunk in a UAF, we created an extra node that points to our target. Let's continue on to the next malloc and break immediately after it.



Malloc recycled the tcache chunk from the head of the freelist. Now only our target chunk remains. One more malloc should do it.



The tcache is empty now.



The malloc returned a pointer to our target and the program wrote to that buffer thinking it was legitimate allocated memory. Instead, it overwrote our target memory that was located elsewhere. If the target were replaced with a function pointer, then control flow hijacking might be possible.

Conclusion

The ptmalloc heap allocator is a good allocator to start the journey of heap exploitation. There is information freely available on its implementation and attacks. The source code is available and debug information can be used out-of-the-box in many OS distributions. TCache poisoning is an interesting attack that takes advantage of heap metadata and highlights one of the things that an attacker might be able to do if they were to corrupt the heap.